AFRL-RI-RS-TR-2008-272
Final Technical Report
October 2008

LARGE SCALE SYSTEM DEFENSE

Columbia University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-272 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

FRANK H. BORN
Work Unit Manager

WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
OCT 2008	Final	Jul 06 – May 08

4. TITLE AND SUBTITLE

LARGE SCALE SYSTEM DEFENSE

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-06-2-0221

5c. PROGRAM ELEMENT NUMBER
31011G

6. AUTHOR(S)

Steven M. Bellovin, Salvatore J. Stolfo and Angelos D. Keromytis

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Columbia University
1700 Broadway
New York NY 10019-5905

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-272

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 08-0325

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The objective of this effort was to investigate techniques for allowing networks composed of many hundreds, thousands, or even millions of commodity computers to protect themselves against a variety of security threats. As a result we developed a number of system prototypes and experimentally demonstrated their effectiveness: an automatic patch generation prototype that can detect previously unknown attacks and create fixes that maintain both integrity and availability of the target application in over 95% of cases with minimal performance overhead; a technique for allowing in situ testing of security patches without affecting the stability or functionality of the production system, using speculative parallel execution; Anagram, a new content-based anomaly detection (AD); Aeolos, a distributed intrusion detection and event correlation infrastructure; STAND, a training-set sanitization technique applicable to ADs requiring unsupervised training; POLYMORPH, an evaluation of the strength of metamorphic engines demonstrating the infeasibility of signature-based filtering devices; and an integrated software diversification system based on Instruction Set Randomization.

15. SUBJECT TERMS
Automatic Patch Generation, Anomaly Detection, Instruction Set Randomization, distributed intrusion detection

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Frank H. Born
U	U	U	UU	25	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Figures

Executive Summary

The objective of this effort was to investigate techniques for allowing networks composed of many hundreds, thousands, or even millions of commodity computers to protect themselves against a variety of security threats. We investigated:

(a) automatic software patch generation for addressing several types of software vulnerabilities and failures;
(b) techniques to enable administrators to quickly roll out vendor-issued software patches without fear of adverse side-effects (*e.g.,* system crashes due to incompatibilities);
(c) anomaly detection (AD) techniques that will allow us to accurately and quickly identify events (attacks, or otherwise anomalous behavior) warranting closer examination; and
(d) techniques for automatically creating diverse instances of software applications such that they are immune to certain types of software vulnerabilities.

We developed a number of system prototypes and experimentally demonstrated their effectiveness:

(a) an automatic patch generation prototype that can detect previously unknown attacks and create fixes that maintain both integrity and availability of the target application in over 95% of cases with minimal performance overhead;
(b) a technique for allowing *in situ* testing of security patches without affecting the stability or functionality of the production system, using speculative parallel execution;
(c) *Anagram*, a new content-based AD; *Aeolos*, a distributed intrusion detection and event correlation infrastructure; *STAND*, a training-set sanitization technique applicable to ADs requiring unsupervised training; *POLYMORPH*, an evaluation of the strength of metamorphic engines demonstrating the infeasibility of signature-based filtering defenses; and
(d) an integrated software diversification system based on Instruction Set Randomization.

Summary of findings

In summary, our significant technical findings in each area are:

(a) Automatic Patch Generation

Our work demonstrated that it is possible to construct a software self-healing mechanism that identifies new instances of known classes of failures (e.g., buffer overflows, input-drive application crashes), creates candidates fixes, tests them in an isolated environments, and (if successful) applies them to the production system. The core intuition behind our work was the notion of *error virtualization*: we treat code blocks (specifically, functions/subroutines) as transactions that may fail. If a failure is observed, we rollback the transaction by reversing the effects of its operation (i.e., undo memory changes and I/O operations) and force program execution to take a different path. Forcing a different path to be taken is achieved by pretending that the failed code block returned with an error code (e.g., "return -1;" in C). Valid error codes to be used, as well as the locations to which program execution needs to be restored, are determined through static analysis of the code and through offline testing/fuzzing. Initial fault detection (first instance of a failure) is determined through use of honeypots or lightweight attack/failure sensors; upon detecting a failure/attack, the system automatically attempts to create and test fixes implementing the transactional processing of the code that failed by adding supervisory code around it. The tests involve replaying the malicious traffic, as well as running a battery of pre-defined tests (software- and site-specific). If a successful fix is found, the administrator is notified; optionally, the system can be configured to roll out the fix immediately, without human supervision. Fixes can be applied both to source code (through automatic code rewriting) and to program binaries.

In addition, we have developed extensions to the Clark-Wilson integrity model to enable integrity recovery based on a policy. This work is intended to provide some assurances with respect to the behavior of the healed application, by specifying specific steps that the supervisory framework has to undertake (or avoid) in the process of recovering from a failure. Currently, such policies must be manually specified by some entity (programmer, administrator, user, ...); the focus of future work should be on automatically determining such policies based on program analysis (whether static or dynamic).

We experimentally determined that program availability and integrity are maintained in over 95% of cases where such fault-handling is performed. The overhead of the rollback mechanism can be as low as 0%, and in all cases we encountered (testing with both real and synthetic bugs/failures) was in the 5% range.

(b) Better Patch Management

Band-aid Patching is a new approach for testing application patches prior to definitive deployment. Using binary runtime injection techniques, we modify binaries so that when program execution reaches a program segment that has been affected (modified) by an issued patch, the program “forks” speculative execution threads that cannot interfere with each other. The output of these threads is examined to discover misbehavior. Our system contains the effects of any misbehavior by unrolling their execution results, and selecting from among the non-misbehaving execution threads. Often, identifying the misbehaving thread is easy: based on its actions (e.g., causing a program crash). Such problems may occur in either the original or the updated version of the code (or,if we are unlucky, in both).

The main advantage of our approach lies in its ability to allow for simple dynamic patches to be tested side-by-side with production-level application state. The output created by these “patch threads” can be then compared, to provide some guarantee over the correctness of the computation. Since there is a cost associated with our approach, our scheme works best when the patches are relatively small in size, or when they affect code that is not in the critical path of the application. Generally, security and stability patches fall in this category. Although our scheme could be applied against any type of patch (e.g., large patches introducing new functionality), the cost and complexity associated would be relatively high.

To validate our approach, we developed a proof-of-concept implementation of Band-aid

Patching using source-to-source transformations and the dyninst runtime instrumentation tool. Specifically, we used dyninst to insert instrumentation trampolines that point to the different versions of the patches under test, and instrumentation that examines and compares the output of these patches. The type of patches that can be applied can range from simple logic bug fixes to testing new vulnerability protection mechanisms. For more complex updates, *i.e.*, where changes to function prototypes and type definitions are required, our approach could be combined with more sophisticated dynamic software update techniques. Our technique uses only the vendor-issued patch and can be applied with little or no intervention by users or administrators. In our work, we investigated some of the issues associated with our approach and identify areas for future investigation.

(c) Artificial Diversity

We developed an ISR-based protection system for network-facing applications. Our scheme represents a hybrid approach to host security that prevents binary code injection attacks. It incorporates three major components: an anomaly-based classifier, a signature-based filtering scheme, and a supervision framework that employs Instruction Set Randomization (ISR). Since ISR prevents code injection attacks and can also precisely identify the injected code, we can tune the classifier and the filter via a learning mechanism based on this feedback. Capturing the injected code allows our system to construct signatures for zero-day exploits. The filter can discard input that is anomalous or matches known malicious input, effectively protecting the application from additional instances of an attack – even zero-day attacks or attacks that are metamorphic in nature. Our system does not require a known user base and can be deployed transparently to clients and with minimal impact on servers. Our prototype protects HTTP servers, but can be applied to a variety of server and client applications.

One of the major contributions of this work is the use of a practical form of ISR. The basic premise of ISR is to prevent code injection attacks from succeeding by creating unique execution environments for individual processes. Code injection is not limited to overflowing stack buffers or format strings. Other injection vectors include web forms that allow arbitrary SQL expressions, CGI scripts that invoke shell programs based on user input, and log files containing character sequences capable of corrupting the terminal display. Our x86 emulator can selectively derandomize portions of an instruction stream, effectively supporting two different instruction sets at the same time. Various processors support the ability to emulate or execute other instruction sets. These abilities could conceivably be leveraged to provide hardware support for ISR. For example, the Transmeta Crusoe chip employs a software layer for interpreting code into its native instruction format. The PowerPC chip employs “Mixed-Mode” execution for supporting the Motorola 68k instruction set. Likewise, the ARM chip can switch freely between executing its regular instruction set and executing the Thumb instruction set. A processor that supports ISR could use a similar capability to switch between executing regular machine instructions and randomized machine instructions. In fact, this is almost exactly what STEM (Selective Transactional Emulation) does in software. Having hardware support for ISR would obviate the need for (along with the performance impact of) software-level ISR.

Our systems receive feedback from the emulator that monitors the execution of a vulnerable application. If the emulator tries to execute injected (non-randomized) code, it catches the fault and notifies the classifier and filter. It can then terminate and restart the process, or simulate an error return from the current function. While our prototype system employs ISR, there are many other types of program supervision that can provide useful information. Each could be employed in parallel to gather as much information as possible. These approaches include input taint tracking, program shepherding, and compiler-inserted checks. One advantage of our system’s feedback mechanism is that it can identify with high confidence the binary code of the attack.

(d) Distributed Anomaly Detection

We developed *Anagram*, a content anomaly detector that models *a mixture of high-order n-grams* ($n > 1$) designed to detect anomalous and “suspicious” network packet payloads. By using higher-order n-grams, Anagram can detect significant anomalous byte sequences and generate robust signatures of validated malicious packet content. The Anagram content models are implemented using highly efficient Bloom filters, reducing space requirements and enabling privacy-preserving cross-site correlation. The sensor models the distinct content flow of a network or host using a semi-supervised training regimen. Previously known exploits, extracted from the signatures of an IDS, are likewise modeled in a Bloom filter and are used during training as well as detection time. We demonstrated that

Anagram can identify anomalous traffic with high accuracy and low false positive rates. Anagram's high-order n-gram analysis technique is also resilient against simple mimicry attacks that blend exploits with "normal" appearing byte padding, such as the blended polymorphic attack recently demonstrated. We also developed *randomized n-gram models*, which further raises the bar and makes it more difficult for attackers to build precise packet structures to evade Anagram even if they know the distribution of the local site content flow. Finally, Anagram's speed and high detection rate makes it valuable not only as a standalone sensor, but also as a network anomaly flow classifier in an instrumented fault-tolerant host-based environment; this enables significant cost amortization and the possibility of a "symbiotic" feedback loop that can improve accuracy and reduce false positive rates over time.

In addition, we developed a technique for more accurately training anomaly detectors (including Anagram). The efficacy of Anomaly Detection (AD) sensors depends heavily on the quality of the data used to train them. Artificial or contrived training data may not provide a realistic view of the deployment environment. Most realistic data sets are dirty; that is, they contain a number of attacks or anomalous events. The size of these high-quality training data sets makes manual removal or labeling of attack data infeasible. As a result, sensors trained on this data can miss attacks and their variations. We extended the training phase of AD sensors (in a manner agnostic to the underlying AD algorithm) to include a sanitization phase. This phase generates multiple models conditioned on small slices of the training data. We used these "micro-models" to produce provisional labels for each training input, and we combined the micro-models in a voting scheme to determine which parts of the training data may represent attacks. Our results suggest that this phase automatically and significantly improves the quality of unlabeled training data by making it as "attack-free" and "regular" as possible in the absence of absolute ground truth. We also showed how a collaborative approach that combines models from different networks or domains can further refine the sanitization process to thwart targeted training or mimicry attacks against a single site.

We also developed Aeolos, a new framework for Collaborative Distributed Intrusion Detection, or CIDS that relies on the combination of a decentralized, robust and scalable P2P architecture, paired with compression algorithms, anonymity and privacy mechanisms to detect attacks accurately and rapidly while encouraging participation. Our insight is the use of a hierarchical distributed hash table (HDHT) which scales well for large-scale alert dissemination; the use of multiple federations of HDHTs to ensure resiliency, provide verifiability and detect uncooperative peers; Bloom filters for effective data privacy and an efficient source of keys for our HDHT; and anonymous-but-differentiable cryptographic signatures to preserve anonymity. Finally, our results show that these techniques are effective for a variety of alert types and rates, ranging from simple IP-based threat determination to more sophisticated payload intrusion detection approaches.

Finally, we developed a *quantitative* analysis of the strengths and limitations of shell-code polymorphism and considered its impact on current intrusion detection practice. We focused on the nature of shell-code *decoding routines*. The empirical evidence we gathered helps show that modeling the *class* of self–modifying code is likely intractable by known methods, including both statistical constructs and string signatures. In addition, we developed measures that provide insight into the capabilities, strengths, and weaknesses of polymorphic engines. In order to explore countermeasures to future polymorphic threats, we showed how to improve polymorphic techniques and create a proof-of-concept engine expressing these improvements. Our results indicate that the class of polymorphic behavior is too greatly spread and varied to model effectively. Our analysis also supplies a novel way to understand the limitations of current signature–based techniques. We conclude that modeling normal content is ultimately a more promising defense mechanism than modeling malicious or abnormal content.

Detailed technical findings

In more detail, our technical findings in each area:

(a) Automatic Patch Generation

We developed ASSURE, a system that provides Automatic Software Self-healing Using REscue points. ASSURE introduces rescue points, to retrofit legacy applications with exception-handling capabilities that mimic system behavior under anticipated error conditions. This behavior is induced to recover from unanticipated software faults while maintaining system integrity and availability. Rescue points are locations in existing application code for handling programmer-anticipated failures which are automatically repurposed and tested for safely enabling general fault recovery. When a fault occurs at an arbitrary location in the program, ASSURE restores execution to the closest rescue point and induces the program to recover execution by virtualizing and using its existing error-handling facilities.

Rescue points virtualize error handling by creating a mapping between the (potentially infinite) set of errors that could occur during a program's execution (e.g., a detected buffer-overflow attack, or an illegal memory-dereference exception) and the limited set of errors that can be handled by the program's code. Thus, a failure that would cause the program to crash is translated into a "return with an error" from an error-handling function along the execution path in which the fault occurred. By reusing existing error-handling facilities and automatically testing them before use in production code, rescue points can reduce the chance of unanticipated execution paths, thereby making recovery more robust. Rescue points do not simply mask errors. Instead, they "teleport" the faults to locations that are known or suspected, with high probability, to handle faults correctly (including correct program-state cleanup).

ASSURE is designed around an Observe Orient Decide Act (OODA) feedback loop. ASSURE first profiles an application offline using fuzzing to identify candidate rescue points. The intuition is that there exists a set of code locations created and tested by the application developers that are routinely used to handle expected errors (*e.g.,* malformed URLs in a web server). ASSURE then uses a set of lightweight software probes to monitor the application during its production use for specific types of faults; a variety of fault monitors can be used. Upon detection of a fault for the first time, ASSURE uses a replica of the application to determine which candidate rescue point can be most effectively used. The selected candidate rescue point is then implemented using exception handling and an operating system checkpoint-restart mechanism that handles multi-process and multi-threaded applications. ASSURE confirms that it has repaired the fault by re-running the application against the event sequence that apparently caused the failure, as well as against already known good and bad input. Upon success, ASSURE uses a runtime binary injection tool to insert the rescue point into the application running on the production server. When the fault occurs again on the production server, the application uses the rescue point to roll back state to the rescue point, where the program is forced to return an error, imitating the behavior observed during fuzzing. The system is designed to operate without human intervention to minimize reaction time, and does not require access to application source code. The end result is automatic short-term healing of software services from what were previously unknown and unforeseen software failures, maintaining both system integrity and availability.

We have implemented an ASSURE Linux prototype that operates without application source code and without base operating system kernel changes. To demonstrate its effectiveness, we have evaluated our prototype on a wide range of real-world server applications and bugs. We focused on server applications because they typically have higher availability requirements and also tend to have short error-propagation distances that lend themselves to our approach. Our experimental results show that ASSURE identified and used rescue points to successfully recover from all of the bugs tested. Unlike other approaches, our evaluation validated ASSURE's ability to recover in the presence of bugs for application deployments in typical multi-process and multi-threaded configurations while running widely used workloads for measuring performance. Our performance measurements showed that ASSURE recovered from faults in just a few milliseconds for all applications and incurred less than 10% performance overhead during normal execution. Furthermore, our results show that ASSURE provides automatic and tested self-healing of

legacy applications in a few seconds to minutes depending on the level of testing desired, orders of magnitude faster than current human-driven patch deployment methods.

While our work in the self-healing space demonstrated the feasibility of the basic concept, these techniques face a few obstacles before they can be deployed to protect and repair legacy systems, production applications, and COTS (Commercial Off–The–Shelf) software. Executing through a fault in this fashion involves overcoming several obstacles. In particular, the semantics of program execution must be maintained as closely as possible to the original intent of the application's author. To that end, we introduced a *repair policy* to guide the semantics of the healing process.

Achieving a semantically correct response remains a key problem for self-healing systems. Executing through a fault or attack involves a certain amount of risk. Even if software could somehow ignore the attack itself, the best sequence of actions leading back to a safe state is an open question. The exploit may have caused a number of changes in state that corrupt execution integrity before an alert is issued. Attempts to self-heal must not only stop an exploit from succeeding or a fault from manifesting, but also repair execution integrity as much as possible. However, self-healing strategies that execute through a fault by effectively pretending it can be handled by the program code or other instrumentation may give rise to semantically incorrect responses. In effect, self-healing may provide a cure worse than the disease. The code snippet below illustrates a specific example: an error may exist in a routine that determines the access control rights for a client. If this fault is exploited, a self-healing technique like error virtualization may return a value that allows the authentication check to succeed. This situation occurs precisely because the recovery mechanism is oblivious to the semantics of the code it protects.

```
int login(UCRED creds)
{
  int authenticated = check_credentials(creds);
  if(authenticated) return login_continue();
  else return login_reject();
}

int check_credentials(UCRED credentials)
{
  strcpy(uname, credentials.username);
  return checkpassword(lookup(uname), credentials);
}
```

One solution to this problem relies on annotating the source code to *(a)* indicate which routines should not be "healed" or *(b)* to provide appropriate return values for such sensitive functions, but we find these techniques unappealing because of the need to modify source code. Since source-level annotations serve as a vestigial policy, we articulated a way to augment self-healing approaches with the notion of *repair policy*. A repair policy (or a recovery policy – we use the terms interchangeably) is specified separately from the source code and describes how execution integrity should be maintained after an attack is detected. Repair policy can provide a way for a user to customize an application's response to an intrusion attempt and can help achieve a completely automated recovery.

We provided a theoretical framework for repair policy by extending the Clark-Wilson Integrity Model (CW) to include the concepts of (a) repair and (b) repair validation. CW is ideally suited to the problem of detecting when constraints on a system's behavior and information structures have been violated. The CW model defined rules that govern three major constructs: constrained data items (CDI), transformation procedures (TP), and integrity verification procedures (IVP). An information system is composed of a set of TPs that transition CDIs from one valid state to another. The system also includes IVPs that measure the integrity of the CDIs at various points of execution.

Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, a vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, repair policy focuses on ways to recover after an unauthorized modification. Our extensions supplements the CW model with primitives and rules for recovering from a policy violation and validating that the recovery was successful.

STEM interprets repair policy to provide a mechanism that can be selectively enforced and retrofitted to the protected application without modifying its source code (although *mapping* constraints to source-level objects assists in maintaining application semantics). As with most self-healing systems, we expect the repairs offered by this "behavior firewall" to be temporary constraints on program behavior — emergency fixes that await a more comprehensive patch from the vendor. One advantage of repair policy is that an administrator can "turn off " a broken repair policy without affecting the execution of the program — unlike a patch.

Repair policy is specified in a file external to the source code of the protected application and is used only by STEM (*i.e.*, the compiler, the linker, and the OS are not involved). This file describes the legal settings for variables in an aborted transaction. The basis of the policy is a list of relations between a transaction and the CDIs that need to be adjusted after error-virtualization, including the return address and return value. A complete repair policy is a wide-ranging topic; in our work, we considered a simple form that:

1. specifies appropriate error virtualization settings to avoid an incorrect return value that causes problems like the one illustrated earlier

2. provides memory rollback for an aborted transaction

3. sets memory locations to particular values

Below we show a sample policy for our running example. The first statement defines a symbolic value. The latter three statements define an IVP, RP, and TP. The IVP defines a simple detector that utilizes STEM's shadow stack. The RP sets the return value to a semantically correct value and indicates that memory changes should be undone, and the TP definition links these measurement and repair activities together. An RP can contain a list of asserted conditions on CDIs that should be true after self-healing completes. The example illustrates the use of the special variable 'rvalue (the apostrophe distinguishes it from any CDI named rvalue). This variable helps customize vanilla error virtualization to avoid problems similar to the one in our code snippet.

```
symval AUTHENTICATION_FAILURE = 0;
ivp MeasureStack :=: ('raddress=='shadowstack[0]);
rp FixAuth :=: ('rvalue==AUTHENTICATION_FAILURE), (rollback);
tp check_credentials
    &MeasureStack :=: &FixAuth;
```

(b) Better Patch Management

Conceptually, our approach can be described by two components: an execution module and a decision module. The execution module provides the speculative execution environment where patches can be applied to service instances. It is possible that multiple such instances run in parallel while being monitored by the execution environment for successful termination or for exception failures. The module provides a recovery mechanism to maintain non-stop service execution in the presence of faults. The decision module consists of a set of detection elements that examine the output (state changes) generated by individual executions, process the results, and reach a decision. The module must be invoked whenever parallel running instances of services reach predefined points in their execution, or when an exception is generated.

The execution module designates the mechanism for the application of patches to service instances. Ideally, the patching mechanism should be able to: *(i)* apply patches to running service instances with little or no down-time, *(ii)* allow for the application of arbitrary patches (i.e., not just binary-compatible), *(iii)* have the ability to apply patches even in the absence of source code, *(iv)* provide the ability to insert multiple patches at specific program locations, and *(v)* monitor multiple speculative executions for successful completion or faults.

There are a few instrumentation injection techniques that can be employed for the purposes of our approach.

• Binary translation, where the tool adds instrumentation to a compiled binary, e.g., ATOM

• Runtime instrumentation, where the code is instrumented just prior to execution, e.g., PIN

• Runtime injection. This is a more light-weight approach than runtime instrumentation, where the code is modified at runtime by inserting jumps to helper functions, e.g., dyninst

Runtime instrumentation provides the most flexible platform, at the cost of higher performance overhead. For example PIN, a runtime injection tool, uses a dynamic translation to intercept and make changes to runtime code, which typically adds a 2X overhead. Dyninst uses a runtime injection approach that adds minimal overhead to the application but requires that patches maintain binary compatibility. The issue of binary compatibility is of minor importance given the fact that the majority of vulnerability patches usually involve minimal changes to the underlying source code. In fact, in order to provide any sort of type-safety on dynamic software updating, one would have to use a language that is dynamic-update aware. The overhead of executing the inserted instrumentation is comparable to that of a function call and primarily depends on how efficiently registers can be saved and restored during execution.

The decision module is responsible for detecting variations, and subsequently violations of execution state. Determining the correctness of the resulting execution is based on a combination of policy and heuristics. Working under the assumption that the base implementation is an instance of "correct" execution, we can derive deviations from this model as potentially problematic. The following two example scenarios illustrate the range of policies that can be used by our approach:

1. Correct base instance: In this context, the assumption is that the base implementation defines the measure of "correctness" for the system. There are numerous ways to determine correctness, one being empirical observation. For example, telephony switches are periodically retrofitted with vulnerability patches that should not alter their normal operation (*e.g.,* how calls are routed). Patches to such systems must characterize functional deviation as potentially problematic.

2. Well defined end-state: In cases where well defined invariants and specifications of the result state are available, execution output can be checked against those constraints. For example, database systems have well-defined data constraints, such as the range of values in fields.

Several mechanisms can be combined as part of our Band-aid Patch architecture. Our goal is to identify differences in state that might occur during different executions and to detect violations to system policy as defined above. The two primary candidates for measuring deviation in execution state are memory traces and I/O transactions. A number of techniques can be applied to memory views generated by the different executions depending on the characteristics of the application and the granularity of information we are trying to extract. Comparing the full memory layout is the simplest approach, but generates copious amounts of information. Context-aware memory "diffs" can be used to filter out superfluous memory information. Memory management at the library/system-call level can be used to fingerprint each execution. Finally, measuring variation in inter/intra–function call-graphs can provide additional hints on the effects of program semantics.

When considering I/O transactions, the basic concept is to fork filesystem views for each execution. At completion time, it is possible to compare file system views for inconsistencies, *e.g.,* using mechanisms such as the versioning file system. Discrepancies and inconsistencies can be handled as indications of anomalous execution. Similarly, for network I/O, operations can be filtered and compared with the anticipated behavior, defined by the appropriate policy or by an external anomaly detector.

The decision component can be injected at specific locations in the code, similar to the patch insertion described earlier. The figure below illustrates the concept. In the example, the execution module is invoked at function foo(), allowing for the simultaneous deployment of two patches. At that time, two more instances of the service are created. Execution continues normally for all instances until we reach *(i)* a predefined point in the execution, or *(ii)* an exception is raised.

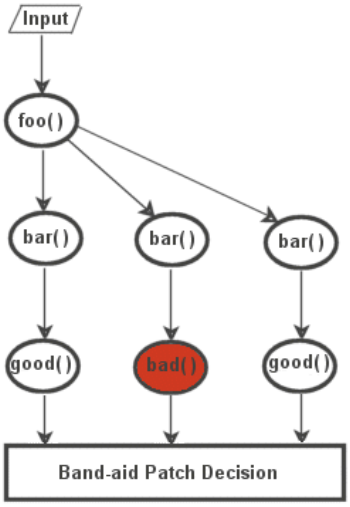

Figure 1 - Band-aid Patch Decision

In our prototype implementation, we chose to use runtime injection for patch insertion. We used dyninst due to its low runtime overhead and its ability to attach and detach from already running processes.

The updates supported by our prototype happen at function-level granularity, and come into effect on the next invocation of the replaced function. The different versions of the patches to be tested are created as a dynamic library that can be linked to the application at runtime. The function where the Band-aid Patching will be initiated in is instrumented to include calls to the patches to be tested on function entry.

Unfortunately, the obvious choice of "forking" a different process for each execution threat carries the stigma associated with changing process information; to avoid breaking program semantics, special care needs to be placed on programs that rely on process ID information. Using dyninst, we can intercept calls to *getpid()* and its derivatives to return the appropriate process ID. However, this solution is no panacea as the process ID might have been communicated to other parts of the application prior to the patch deployment. A possible solution is to add a thin virtualization layer that maps all process IDs to virtual values. An alternative approach would be to execute the different versions of a patch in succession. While this approach is elegant in its simplicity, it means that we can only explore the effects of a patch within the confines of the function, *i.e.,* we would not be able to examine possible side-effects exhibited by a patch farther down in program execution. We also cannot take advantage of hardware facilities, such as multiple processors/cores, that could minimize the overhead of our scheme. For our particular implementation we employ the sequential patch approach using a tool we have previously developed, STEM. The decision component is currently implemented as an application-specific component that can be injected at particular program locations using the mechanism described previously. For our prototype, we used STEM to detect general faults and memory violations. If a patch does not introduce a failure during execution, execution is allowed to continue. If the patch causes a failure, execution continues with the unpatched version of the code.

(c) Artificial Diversity

The design of our system is based on two major components: a filtering proxy and an application supervision framework. A major goal of the design is to keep the system modular and deployable on a single host. The figure below shows a high-level view of this design. The protected application can be either a server waiting for requests or a client program receiving input. Input to a client program or requests to a server are passed through the filtering proxy and dropped if deemed malicious. If the supervision framework detects something wrong with the protected application, it signals the filter to update its signatures and models. Although server replies and outgoing client traffic can also be modeled and filtered, our current implementation does not perform this extra step. Outgoing filtering is useful in protecting a client application by stopping information leaks or the spread of self-propagating malware.

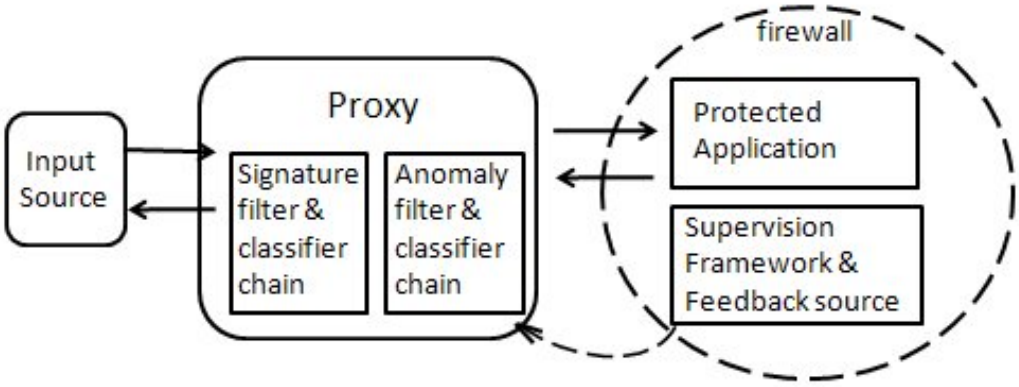

Figure 1 - General Architecture of Feedback Learning Intrusion System (FLIPS)

The function of the proxy is to grade or score the input and optionally drop it. The proxy is a hybrid of the two major classification schemes, and its subcomponents reflect this dichotomy. A chain of signature-based filters can score and drop a request if it matches known malicious data, and a chain of anomaly-based classifiers can score and drop the input if it is outside the normal model. Either chain can allow the request to pass even if it is anomalous or matches previous malicious input. The default policy for our prototype implementation is to only drop requests that match a signature filter. Requests that the anomaly classifier deems suspicious are copied to a cache and forwarded on to the application. We adopt this stance to avoid dropping requests that the anomaly component mislabels (false positives). The current implementation only drops requests that have been confirmed to be malicious to the protected application and requests that are closely related to such inputs.

The function of the application supervision framework is to provide a way to stop an exploit, automatically repair the exploited vulnerability, and report information about an exploit back to the filters and classifiers. Similar to the filtering and classification chains, the supervision framework could include a number of host-based monitors to provide a wide array of complementary feedback information to the proxy. Our prototype implementation is based on one type of monitor (ISR) and will only provide feedback information related to code-injection attacks. Many other types of attacks are possible, and whether something is an attack or not often depends on context. Our design allows for an array of more complicated monitors. STEM allows the application to recover from a code injection attack by simulating an error return from the emulated function after notifying the proxy about the injected code.

We assume a threat model that closely matches that of previous ISR efforts. Specifically, we assume that an attacker does not have access to the randomized binary or the key used to effect achieve this randomization. These objects are usually stored on a system's disk or in system memory; we assume the attacker does not have local access to these resources. In addition, the attacker's intent is to inject code into a running process and thereby gain control over the process by virtue of the injected instructions. ISR is especially effective against these types of threats because it interferes with an attacker's ability to automate the attack. The entire target population executes binaries encoded under keys unique to each instance. A successful breach on one machine does not weaken the security of other target hosts.

While the design of our system is quite flexible, the nature of host-based protection and our choices for a prototype implementation impose several limitations. First, host-based protection mechanisms are thought to be difficult to manage because of the potential scale of large deployments. Outside the enterprise environment, home users are unlikely to have the technical skill to monitor and patch a complicated system. We purposefully designed our system to require little management beyond installation and initial training. PayL (payload anomaly detector) can perform unsupervised training. One task that should be performed during system installation is the addition of a firewall rule that redirects traffic aimed at the protected application to the proxy and only allows the proxy to contact the protected application. Second, the performance of such a system is an important consideration in deployment. If the cost is deemed too high, the system can still be deployed as a honeypot or a "twin system" that receives a copy of input meant for another host. Third, the proxy should be as simple as possible to promote confidence in its code-base

that it is not susceptible to the same exploits as the protected application. We implement our proxy in Java, a type-safe language that is not vulnerable to the same set of binary code injection attacks as a C program. Our current implementation only considers HTTP request lines. Specifically, it does not train or detect on headers or HTTP entity bodies. Therefore, it only protects against binary code injection attacks contained in the request line. However, nothing prevents the scope of training and detecting from being expanded, and other types of attacks can be detected at the host.

Our supervision framework is an application-level library that provides an emulator capable of switching freely between de-randomizing the instruction stream and normal execution of the instruction stream on the underlying hardware.

STEM is an x86 emulator that can be selectively invoked for arbitrary code segments, allowing us to mix emulated and non-emulated execution inside the same process. The emulator lets us *(a)* monitor for de-randomization failures when executing the instruction, *(b)* undo any memory changes made by the code function inside which the fault occurred, and *(c)* simulate an error return from said function (*i.e.,* the *error virtualization* technique developed as part of the Automatic Patch Generation thrust of this project). One of our key assumptions is that we can create a mapping between the set of errors and exceptions that could occur during a program's execution and the limited set of errors that are explicitly handled by the program's code.

The main loop of the emulator fetches, decodes, executes, and retires one instruction at a time. Before fetching an instruction, de-randomization takes place. Since the x86 architecture contains variable-length instructions, translating enough bytes in the instruction stream is vital for the success of decoding. Otherwise, invalid operations may be generated. To simplify the problem, we assume the maximum length (16 bytes) for every instruction. For every iteration of the loop, 16-bit words are XOR'd with a 16-bit key and copied to a buffer. The fetch/decode function reads the buffer and extracts one instruction. The program counter is incremented by the exact length of the processed instruction. In cases where instructions are fifteen bytes or less, unnecessary de-randomization takes place, but this is an unavoidable side-effect of variable-length instructions. If injected code resides anywhere along the execution path, the XOR function will convert it to an illegal opcode or an instruction which will access an invalid memory address. If an exception occurs during emulation, STEM notifies the proxy of the code at the instruction pointer. STEM captures 1KB of code and opens a simple TCP socket to the proxy (the address and port of the feedback mechanism are included in the startup options). STEM then simulates an error return from the function it was invoked in.

We evaluated the performance impact of STEM by instrumenting the Apache web server and performing micro-benchmarks on some shell utilities. We chose the Apache flood_httpd testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in the figure below. The value for total number of requests per second is extrapolated (by flood 's reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not be interpreted to mean that our Apache instances actually served some 6,000 requests per second.

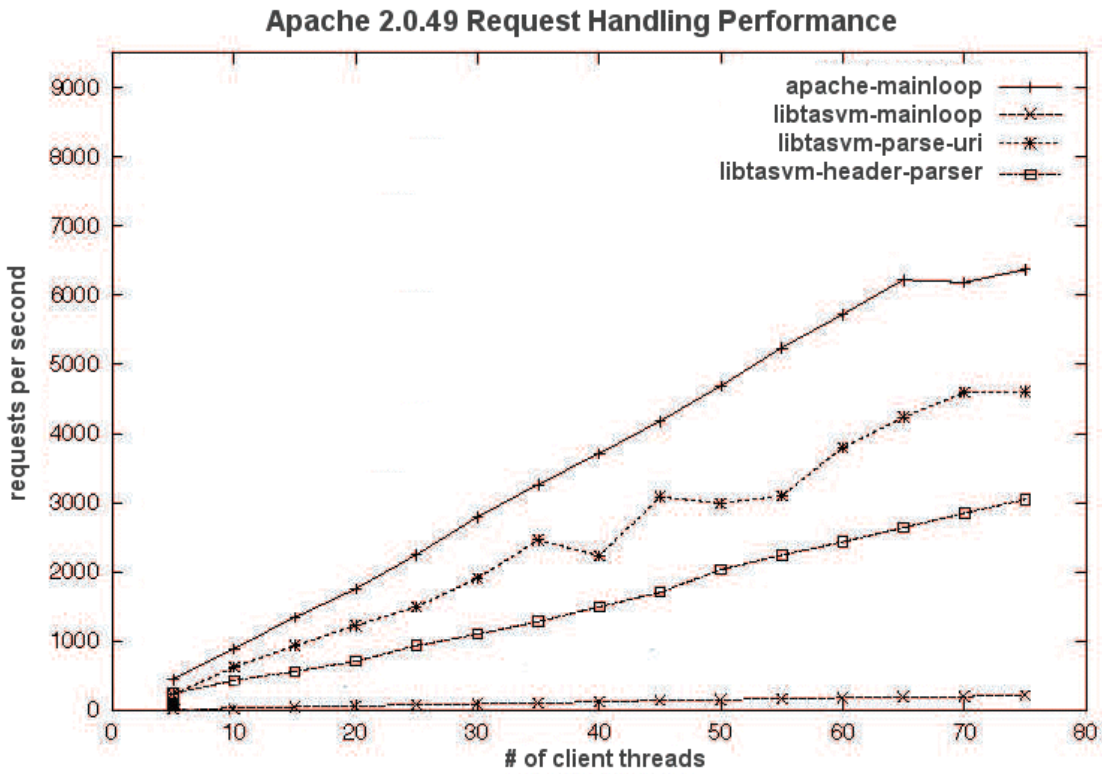

Figure 3 - Performance Of The System Under Various Levels Of Emulation

We selected some common shell utilities and measured their performance for large workloads running both with and without STEM. For example, we issued an '*ls -R*' command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). The following table shows the result of these measurements. As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.

Test Type	Trials	Mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.8	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

Figure 4 - Microbenchmark Performance Times For Various Command Line Utilities

To demonstrate the operation of the system, we inserted a synthetic code injection vulnerability into Apache. The vulnerability was a simple stack-based overflow of a local fixed-size buffer. The function was protected with STEM, and we observed how long it took our system to stop the attack and deploy a filter against further instances.

To test the end-to-end functionality, we directed two streams of attack instances against Apache through our proxy. We first sent a stream of 67 identical attack instances and then followed this with 22 more attacks that included slight variations of the original attack. In the first attack stream, our system successfully blocked 61 of the 67 attack

instances. It let the first six instances through before STEM had enough time to feedback to our system. It took roughly one second for our system to start blocking the attacks. After that, each subsequent identical attack instance was blocked by the direct match filter. The second attack stream contained 22 variations of the original. The LCS filter (with a threshold of 60%) successfully blocked twenty of these. This result provides some evidence that our approach can stop metamorphic attacks.

(d) Distributed Anomaly Detection

ANAGRAM: Anagram introduced the use of Bloom filters and a binary-based detection model. Anagram does not compute frequency distributions of normal content flows; instead, it trains its model by storing all of the distinct n-grams observed during training in a Bloom filter without counting the occurrences of these n-grams. Anagram also stores n-grams extracted from known malicious packets in a second *bad content Bloom filter*, acquired by extracting n-grams from openly available worm detection rules, such as the latest Snort rulesets. At detection time, packets are scored by the sensor on the basis of the number of unobserved n-grams the packet contains. The score is weighted by the number of malicious n-grams it contains as well. We demonstrated that this semi-supervised strategy attains remarkably high detection and low false positive rates, in some cases 100% detection with less than 0.006% false positive rate (per packet).

The use of Bloom filters makes Anagram memory and computationally efficient and allows for the modeling of a *mixture of different sizes of n-grams* extracted from packet payloads, i.e., an Anagram model need not contain samples of a fixed size gram. This strategy is demonstrated to exceed PAYL (our previous AD) in both detection and false positives rates. Furthermore, Anagram's modeling technique is easier to train, and allows for the estimation of when the sensor has been trained enough for deployment. The Bloom filter model representation also provides the added benefit of preserving the privacy of shared content models and alerts for cross-site correlation.

Of particular interest is that Anagram is shown to be robust against existing mimicry attack approaches. We demonstrated Anagram's ability to counter the simple mimicry attacks levied against PAYL. Furthermore, Anagram is designed to defeat training and mimicry attacks by using *randomized n-gram modeling*. The approach presented raises the bar against the enemy, making it far more difficult to design an n-gram attack against Anagram. By randomizing the portion of packets that Anagram extracts and models, mimicry attackers cannot easily guess how and where to pad malicious content to avoid detection. We also made use of a feedback loop between the Anagram sensor and host-based detectors, thereby updating Anagram models over time and improving its detection performance. Thus, the combination of model randomization and a feedback loop makes it more difficult to evade detection by training and mimicry attacks.

While higher order n-grams contain more information about payloads, the feature space grows exponentially as n increases. Comparing an n-gram frequency distribution against a model is infeasible since the training data is simply too sparse; the length of a packet is too small compared to the total feature space size of a higher-order n-gram. One TCP packet may contain only a thousand or so n-grams, while the feature space size is 256^n. Clearly, with increasing n, generating sufficient frequency statistics to estimate the true empirical distribution accurately is simply not possible in a reasonable amount of time. In Anagram, we therefore do not model the frequency distribution of each n-gram. Rather, we observe each distinct n-gram seen in the training data and record each in a space efficient Bloom filter. Once the training phase terminates, each packet is scored by measuring the number of n-grams it did not observe in the training phase. Hence, a packet is scored by the following formula, where *Nnew* is the number of new n-grams not seen before and T is the total number of n-grams in the packet: $Score = Nnew / T \in [0,1]$. At first glance, the frequency-based n-gram distribution may contain more information about packet content; one might suspect it would model data more accurately and perform better at detecting anomalous data, but since the training data is sparse, this alternative "binary-based model" performs significantly better than the frequency-based approach given the same amount of training data.

We analyzed the network traffic for the Columbia Computer Science website and, as expected, a small portion of the n-grams appear frequently while there is a long "tail" of n-grams that appear very infrequently. Since a significant number of n-grams have a small frequency count, and the number of n-grams in a packet is very small relative to the whole feature space, the frequency-distribution model incurs relatively high false positives. Thus, the binary-based model (simply recording the distinct n-grams seen during training) provides a reasonable estimate of how "normal"

a packet may be. This is a rather surprising observation, and works very well in practice. The conjecture is that true attacks will be delivered in packets that contain many more n-grams not observed in training than "normal" packets used to train the model. After all, a true zero-day attack must deliver data to a server application that has never been processed by that application before. Hence, the data exercising the vulnerability is very likely to be an n-gram of some size never before observed. By modeling a mixture of n-grams, we increase the likelihood of observing these anomalous grams.

To validate this conjecture, we compared the ROC curves of the frequency-based approach and the binary-based approach for the same datasets (representing equivalent training times). We collected the web traffic of two CS departmental web servers, *www* and *www1*; the former serves the department webpage, while the latter serves personal web pages. Traffic was collected for two different time periods: a period of sniffed traffic from the year 2004 and another dataset sniffed in 2006. The 2004 datasets (*www-04* and *www1-04*) contain 160 hours of traffic; the 2006 datasets (*www-06* and *www1-06*) contain about 560 hours. We tested for the detection of several real worms and viruses: CodeRed, CodeRed II, WebDAV, Mirela, a php forum attack, and a worm that exploits the IIS Windows media service, the nsiislog.dll buffer overflow vulnerability (MS03-022). These worm samples were collected from real traffic as they appeared in the wild, from both our own dataset and from a third-party.

For the first experiment, we used 90 hours of www1-06 for training and 72 hours for testing. (Similar experiments on the other datasets display similar results, and we skip them here for brevity.) Both the detection rate and false positive rate are calculated based on packets with payloads; non-payload (e.g., control) packets were ignored. Some attacks have multiple packets; while fragmentation can result in a few packets appearing normal, we can still guarantee reliable attack detection over the entire set of packets. For example, for the IIS5 WebDAV attack, 5-grams detect 24 out of 25 packets as being anomalous. The only missed packet is the first packet, which contains the buffer overflow string "SEARCH /AAA……AA"; this is not the key exploit part of the attack.

The binary-based approach yields significantly better results than the frequency-based approach. When a 100% detection rate is achieved for the packet traces analyzed, the false positive rate of the binary-based approach is at least *one order of magnitude less* than the frequency-based approach. The relatively high false positive rate of the frequency-based approach suggests much more training is needed to capture accurate statistical information to be competitive. In addition, the extremely high false positive rate of the 3-gram frequency-based approach is due to the fact that the 3-grams of the php attack all appear frequently enough to make it hard to distinguish them from normal content packets. On the other hand, the binary-based approach used in Anagram results in far better performance. The 0.01% false positives average to about 1 alert per hour for *www1* and about 0.6 alerts per hour for *www*. The result also shows that 5-grams and 6-grams give the best result, and we've found this to be true for others as well.

As previously stated, Anagram may easily model a mixture of different n-grams simply by storing these in the same Bloom filter. However, for larger n-grams additional training may be required; our modeling approach allows us to estimate when the sensor has been sufficiently trained.

One significant issue when modeling with higher order n-grams is memory overhead. By leveraging the binary-based approach, we can use more memory-efficient set-based data structures to represent the set of observed n-grams. In particular, the *Bloom filter* (BF) is a convenient tool to represent the binary model. Instead of using n bytes to represent the n-gram, or even 4 bytes for a 32-bit hash of the n-gram, the Bloom filter can represent a set entry with just a few bits, reducing memory requirements by an order of magnitude or more.

A Bloom filter is essentially a bit array of m bits, where any individual bit i is set if the hash of an input value, mod m, is i. As with a hash table, a Bloom filter acts as a convenient one-way data structure that can contain many items, but generally is orders- of-magnitude smaller. Operations on a Bloom filter are $O(1)$, keeping computational overhead low. A Bloom filter contains no false negatives, but may contain false positives if collisions occur; the false positive rate can be optimized by changing the size of the bit array and by using multiple hash functions (and requiring all of them to be set for an item to be verified as present in the Bloom filter; in the rare case where one hash function collides between two elements, it's highly unlikely a second or a third would also simultaneously collide). By using universal hash functions, we can minimize the probability of multiple collisions for n-grams in one packet (assuming each n-gram is statistically independent); the Bloom filter is therefore safe to use and does not negatively affect detection accuracy.

STAND: Ideally, an anomaly detector should achieve 100% detection accuracy, *i.e.,* true attacks are all identified, with 0% false positives. Reaching this ideal is very hard due to a number of problems. First, the generated model can under-fit the actual normal traffic. Under-fitting means that the AD sensor is overly general: it will flag traffic as "normal" even if this traffic does not belong to the true normal model. As a result, attackers have sufficient space to disguise their exploit, thus increasing the amount of "false negatives" produced by the sensor. Second, and equally as troubling, the model of normal traffic can over-fit the training data: non-attack traffic that is not observed during training may be regarded as anomalous. Over-fitting can generate an excessive amount of false alerts or "false positives." Third, unsupervised AD systems often lack a measure of ground truth to compare to and verify against. The presence of an attack in the training data "poisons" the normal model, thus rendering the AD system incapable of detecting future or closely related instances of this attack. As a result, the AD system may produce false negatives. This risk becomes a limiting factor of the size of the training set. Finally, even in the presence of ground truth, creating a single model of normal traffic that includes all non-attack traffic can result in under-fitting and over-generalization.

These problems appear to stem from a common source: the quality of the normality model that an AD system employs to detect abnormal traffic. This single, monolithic normality model is the product of a training phase that traditionally uses all the traffic from a non-sanitized training data set. Our goal in our work was to extend the AD training phase to successfully sanitize training data by removing both attacks and non-regular traffic, thereby computing a more accurate anomaly detection model that achieves both a high rate of detection and a low rate of false positives. To that end, we generalized the notion of training for an AD system. Instead of using a normal model generated by a single AD sensor trained on a single large set of data, we use multiple AD instances trained on small data slices. This process produces multiple normal models, which we call *micro-models*, by training AD instances on small, disjoint subsets of the original traffic dataset. Each of these micro- models represents a very localized view of the training data. Armed with the micro-models, we are now in a position to assess the quality of our training data and automatically detect and remove any attacks or abnormalities that should not be considered part of the normal model.

The intuition behind our approach is based on the observation that in a training set spanning a sufficiently large time interval, an attack or an abnormality will appear only in small and relatively confined time intervals. To identify these abnormalities, we test each packet of the training data set against the produced micro-models. Using a voting scheme, we can determine which packets to consider abnormal and remove from our training set. In our analysis, we explored the efficiency and tradeoffs of both majority voting and weighted voting schemes. The result of our approach is a training set which contains packets that are closer to what we consider the "normal model" of the application's I/O streams.

This sanitized training set enables us to generate a single *sanitized model* from a single AD instance. This model is very likely free of both attacks and abnormalities. As a result, the detection performance during the testing phase should improve. Our experimental results show a 5-fold increase of the average detection rate. Furthermore, data that was deemed abnormal in the voting strategy is used for building a different model, which we call the *abnormal model*. This model is intended to represent traffic that contains attacks or any data that is not commonly seen during a normal execution of the protected system.

Our initial assumptions do not hold when the training set contains persistent and/or targeted attacks, or there exist other anomalies that persist throughout the majority of the training set. To defend against such attacks, we proposed a novel, fully distributed collaborative sanitization strategy. This strategy leverages the location diversity of collaborating sites to exchange information related to abnormal data that can be used to clean each site's training data set. Consequently, our work introduces a two-phase training process: initially, we compute the AD models of "normal" and "abnormal" locally from the training set at each site. In the second phase, we distribute the "abnormal" models between sites, and we use this information to re-evaluate and filter the local training data set. If data deemed normal by the local micro-models happens to belong to a remote "abnormal" model, we inspect or redirect this data to an oracle. Even if the identities of the collaborating sites become known, attacking all the sites with targeted or blending attacks is a challenging task. The attacker will have to generate mimicry attacks against all collaborators and blend the attack traffic using the individual sites' normal data models.

Our evaluation considered two different defense configurations involving AD sensors. In the first case, we measured the increase in detection performance for a simple AD-based defense system when we use the new training phase to

sanitize the training set. As a second scenario, we assumed that a latency-expensive oracle can help classify "suspect data" and differentiate between false positives (FP) and true positives (TP). In practice, our oracle consists of a heavily instrumented host-based "shadow" server system that determines with very high accuracy whether a packet contains an attack. By diverting all suspect data to this oracle, we can identify true attacks by observing whether the shadow sensor emits an alert after consuming suspicious data. This high accuracy, however, comes at the cost of greatly increased computational effort making the redirection of all traffic to the shadow sensors infeasible.

Many papers comment on anomaly detectors having too high a false positive rate, thus making them less than ideal sensors. In light of the above scenario, we see such comments as the "*false* false positive problem," as our shadow sensor architecture allows an automated process (instead of a human operator) to consume and vet FPs. We used this scenario to demonstrate that failure to substantially reduce the FP rate of a network AD sensor does not render the sensor useless. By using a host-based shadow sensor, false positives neither damage the system under protection nor flood an operational center with alarms. Instead, the shadow sensor processes both true attacks and incorrectly classified packets to validate whether a packet signifies a true attack. These packets are still processed by the shadowed application and only cause an increased delay for network traffic incorrectly deemed an attack.

Distributed Network Anomaly Detection (DNAD): Existing "distributed" intrusion detection systems largely suffer from two fundamental flaws. First, they do not address scalability issues in a meaningful fashion. Most systems are either completely centralized, making them prone to denial-of-service attacks or reliability issues, or completely decentralized, making effective alert dissemination problematic as significant replication is needed, increasing the latency and decreasing the reliability of alert exchange. Second, they do not adequately address privacy requirements. Organizations are reluctant to collaborate against threats when sensitive network topology or network traffic could be exchanged. The few existing approaches that address this problem do so by largely scrubbing data and making peers completely anonymous. This approach is inadequate, because it eliminates meaningful data correlation and allows for little or no resiliency to unscrupulous collaborators, who may flood the network with noise to try and reduce the effectiveness of cross-organization collaboration. Here, we describe a new Collaborative Intrusion Detection System (CIDS) architecture that works with existing single-enclave IDS sensors and implements an effective distribution, aggregation, and correlation layer to existing IDS sensors to address both of these problems, enabling a feasible intrusion alert-sharing infrastructure for the first time. The aspects to this architecture include the following:

1) We employ Hierarchical Distributed Hash Tables, or HDHTs, that aim to provide a decentralized alert dissemination infrastructure with the introduction of node hierarchies to enable more effective scaling. The key differentiating concept to enabling HDHTs for intrusion detection is the creation and partitioning of intrusion alert keys to determine both "cluster" and "node" identifiers, where the node is local to the aforementioned cluster. This enables broad decentralization, but also avoids building a very large, flat network, which dramatically increases the latency of alert dissemination. We also propose the use of multiple hierarchies with a collaborating network, *i.e.,* all nodes are involved in multiple federations, each with different node topology, and disseminate alerts in each federation. This provides resiliency in the case of network failure or partitioning, and provides an inherent verification mechanism to determine erratic or uncooperative nodes who may be trying to subvert the collaboration network by (selectively or completely) withholding information as it is routed. HDHTs also naturally enable the use of hierarchical aggregation tools, which can be used to further reduce alert rates and, correspondingly, communication overhead.

2) We include the notion of privacy-preserving transforms to convert potentially sensitive alert information into intermediate structures that offer effective correlation without yielding secrets to other honest-but-curious or malicious participants. These transforms, based on hashing principles and use of Bloom filters, are fast to generate and compact, enabling quick message exchange when large alert volumes are generated.

3) The architecture supports the use of signatures for accountability, enabling what we term anonymous differentiability—that is, while the actual identity of participants should not be revealed, nodes should be able to effectively distinguish between alert sources and effectively decide whether to trust a particular source or not. These signatures are employed in an Onion routing-style mechanism to ensure that intermediate nodes in the decentralized framework do not pollute data that are replicated throughout the network.

The inclusion of these three key features enables a comprehensive platform for effective Collaborative Intrusion Detection.

POLYMORPH: Conventional wisdom has held that attackers retain a significant advantage by using polymorphic tactics to disguise their shell-code. To the best of our knowledge, however, there exists no quantitative analysis of this advantage. Our work in this space provides empirical evidence to support this folk wisdom and helps improve understanding of the polymorphic shell-code problem in the following ways:

• We illustrate the ultimate futility of string–based signature schemes by showing that the class of n-byte decoder samples spans n-space. Although our results should not be interpreted as a call for the immediate abandonment of all signature–based techniques, we believe there is a strong case for investigating other protection paradigms.

• As a corollary, we showed that given any normal statistical model, there is a significant probability that an attacker can craft successful targeted attacks against it.

• We proposed metrics to gauge the relative strengths of polymorphic engines, and we use these to examine some of the current state-of-the-art engines. We believe our methodology is novel and helps provide some insight in a space that has generally been lacking in quantitative analysis.

• We showed how to augment existing polymorphic engines and demonstrate this process by presenting our implementation of a proof-of-concept engine.

The problem we address can be stated as follows:

PROBLEM DEFINITION *Given* n *bytes, there can be a set of* 256n *possible strings. The specific class of x86 code of length* n *that corresponds to decoders is a subset of this superset, i.e., spanning a subspace within this larger space. How difficult is it to model this subspace — in other words, what is the magnitude of this span? What are polymorphic threats we can expect to see in the immediate future? Finally, what are the theoretical limits?*

We combined a number of methods to answer this question. First, we introduced a set of measures to assess the randomness of a population of samples. We employed these measures to analyze a pool of decoders generated by existing polymorphic engines. Next, we demonstrated our improvements to existing polymorphic techniques. Finally, we analyzed the theoretical limits of polymorphism by examining a fixed-size space of n-bytes. We explored this space using efficient genetic algorithms to characterize the span of all x86 code that exhibits polymorphic behavior. Along the way, we explained why signature-based detection currently works, why it may work in the short term, and why it will progressively become less valuable. We also discovered that shell-code behavior varies enough to not only present a challenge for signature systems, but also presents a significant challenge for statistical approaches to model malcode.

Our results show that the span of polymorphic code likely reaches across n-space. The challenge of signature–based detection is to model a space on the order of $O(2^{8 \cdot n})$ signatures to catch potential attacks hidden by polymorphism. To cover thirty-byte decoders requires $O(2^{240})$ potential signatures; for comparison there exist an estimated 2^{80} atoms in the universe. We would much sooner run out of atoms than attackers run out of decoders. Current signature schemes work only because of advances in rapid isolation and generation of signatures. This strategy may work for the short term; however, our work indicates that defenders cannot capture the initiative from the attacker under this reactive defense strategy. Somewhat troubling is the additional implication that regardless of what a normal model of traffic for a particular site may be, there exists a certain probability that a range of decoders would fall within the span of that normal model because sequences which exhibit polymorphic behavior span most of n-space.

Appendix A: Deliverables (software, papers, theses, talks)

Theses

"Integrity Postures for Software Self-Defense"
Michael E. Locasto, Dept. of Computer Science, Columbia University, December 2007

"Software Self-healing Using Error Virtualization"
Stelios Sidiroglou, Dept. of Computer Science, Columbia University, May 2008

Conference Publications

"Casting out Demons: Sanitizing Training Data for Anomaly Sensors"
Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. In Proceedings of the IEEE Symposium on Security & Privacy, pp. 81 - 95. May 2008, Oakland, CA. (Acceptance Rate: 11.2%)

"On the Infeasibility of Modeling Polymorphic Shellcode"
Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), pp. 541 - 551. October/November 2007, Alexandria, VA. Also available as Columbia University Computer Science Department Technical Report CUCS-007-07. (Acceptance rate: 18.1%)

"Defending Against Next Generation Attacks Through Network/Endpoint Collaboration and Interaction"
Spiros Antonatos, Michael E. Locasto, Stelios Sidiroglou, Angelos D. Keromytis, and Evangelos Markatos. In Proceedings of the 3rd European Conference on Computer Network Defense (EC2ND). October 2007, Heraclion, Greece. (Invited paper)

"Characterizing Self-healing Software Systems"
Angelos D. Keromytis. In Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS), pp. 22 - 33. September 2007, St. Petersburg, Russia. (Invited paper)

"From STEM to SEAD: Speculative Execution for Automated Defense"
Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, and Angelos D. Keromytis. In Proceedings of the USENIX Annual Technical Conference, pp. 219 - 232. June 2007, Santa Clara, CA. An older version is available as Columbia University Computer Science Department Technical Report CUCS-004-07. (Acceptance rate: 18.75%)

"Using Rescue Points to Navigate Software Recovery (Short Paper)"
Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh. In Proceedings of the IEEE Symposium on Security & Privacy, pp. 273 - 278. May 2007, Oakland, CA. (Acceptance rate: 8.3%)

"Poster Paper: Band-aid Patching"
Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis. In Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep), pp. 102 - 106. June 2007, Edinburgh, UK.

"Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems"
Gabriela F. Cretu, Angelos Stavrou, Salvatore J. Stolfo, and Angelos D. Keromytis. In

Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep), pp. 64 - 70. June 2007, Edinburgh, UK.

"Next Generation Attacks on the Internet"
Evangelos Markatos and Angelos D. Keromytis. In Proceedings (electronic) of the EU-US Summit Series on Cyber Trust: Workshop on System Dependability & Security, pp. 67 - 73. November 2006, Dublin, Ireland. (Invited paper)

"A Model for Automatically Repairing Execution Integrity"
Michael E. Locasto, Gabriela F. Cretu, Angelos Stavrou, and Angelos D. Keromytis. Columbia University Computer Science Department Technical Report CUCS-005-07, January 2007.

"Speculative Execution as an Operating System Service"
Michael E. Locasto and Angelos D. Keromytis. Columbia University Computer Science Department Technical Report CUCS-024-06, May 2006.

"Quantifying Application Behavior Space for Detection and Self-Healing"
Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis, and Salvatore J. Stolfo. Columbia University Computer Science Department Technical Report CUCS-017-06, April 2006.

"Bloodhound: Searching Out Malicious Input in Network Flows for Automatic Repair Validation"
Michael E. Locasto, Matthew Burnside, and Angelos D. Keromytis. Columbia University Computer Science Department Technical Report CUCS-016-06, April 2006.

Talks (other than those associated with the papers listed above)

"Self-healing Software Systems", Computer Science Department, Athens University of Economics and Business (AUEB), Athens, Greece, May 2008.

"Using Instruction Set Randomization, and its Limitations", AFOSR Invitational Workshop on Homogeneous Enclave Software vs. Heterogeneous Enclave Software, October 2007.

"Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems", Institute of Computer Science (ICS), Foundation of Research and Technology Hellas (FORTH), July 2007.

"Application Communities: A Collaborative Approach To Software Security", IBM Research, July 2007.

"Application Communities: A Collaborative Approach To Software Security," CS Department, Athens University of Economics and Business (AUEB), April 2007

"Self-Healing Software", National Institute of Advanced Industrial Science and Technology (AIST), Japan, April 2007.
"Toward Self-healing Software," Institute for Infocomm Research, Singapore, March 2007

"Application Communities: A Collaborative Approach To Software Security," CS Department, Stony Brook University, Stony Brook NY, November 2006

"Using Virtual Machines to Combat Malware", National Security Research Institute (NSRI), Republic of Korea, November 2006.

"Application Communities: A Collaborative Approach to Software Security", Computer Science Department, Johns Hopkins University, October 2006.

"Application Communities: A Collaborative Approach to Software Security", Computer Science Department, University of Toronto, September 2006.

"Application Communities: A Collaborative Approach To Software Security," CS Department, Brooklyn Polytechnic, New York, September 2006

"Application Communities: A Collaborative Approach to Software Security", Computer Science Department, University of Crete, July 2006.

"Application Communities," Institute for Infocomm Research, Singapore, June 2006

Software

- Automatic Patch Generation prototype

  This prototype implements the source-code-based APG system, including a web-based GUI. (Delivered in the form of a USB hard drive containing virtual machine images configured to run the software.)

- FLIPS

  This prototype implements the artificial software diversification system using Instruction Set Randomization (ISR) through a selective code-slice emulator. In addition, the system includes a signature generator and an anomaly detector that is integrated with the ISR system to create signatures and train AD models for attack characterization.

- Anagram

  This prototype implements our Bloom Filter-based anomaly detector. (Delivered to the NSA, contact person: Ray Roy.)

- STAND

  This prototype implements our anomaly detector training-set sanitization scheme. (Delivered to the NSA, contact person: Ray Roy.)

- Better Patch Management

  This prototype implements our system for in-situ testing of security patches through dynamic replica creation and management.